

第13章动态对象创建

本章内容

- 动态对象的创建和清理是大多数程序设计语言的核心内容之一。
- C++中，如何保证正确的初始化和清理，同时在堆上获取内存。

对象创建

- 创建一个C++对象时，会发生两件事：
 - 为对象分配内存
 - 调用构造函数对内存进行初始化

对象的创建

- 有三种方式为对象分配内存
 - 在静态存储区域，存储空间在程序开始之前就可以分配。这个存储空间在程序的整个运行期间都存在。
 - 无论何时，到达一个特殊的执行点：块的入口，定义在该块中的存储单元在栈上被创建。离开块的时候，这个存储单元被自动释放。
 - 存储单元也可以从一块称为堆的地方分配，这被称为动态内存分配。当然也需要负责决定何时释放内存。

动态内存分配：从C语言到C++

- C语言：申请/释放内存
 - 为了在运行时动态分配内存，C在它的标准库函数中提供了一些函数在堆中分配内存：
 - `void * malloc(int n);`
 - `void * calloc(int n,int size);`
 - `void free(void *p)`
 - `void * realloc(void * p,int n);`

动态内存分配：从C语言到C++

- C++：创建对象（new）
 - 把创建一个对象所需的所有动作都结合在一个称为new的运算符中。当用new创建一个对象时，他就在堆里为对象分配内存，并为这块内存调用构造函数
 - `MyType *fp = new MyType(1,2);`
 - 上述的操作可以分解为两个步骤：
 - 先调用`malloc(sizeof(MyType))`
 - 然后调用以1， 2为参数的构造函数为对象初始化对象。

动态内存分配：从C语言到C++

- C++：删除对象（delete）
 - 删除对象通常意味着释放内存，但往往也伴随着一些其它的操作，例如其他资源的释放。
 - 对于上面用new申请的对象，删除的方式是：
 - delete fp;
 - 上述代码的效果是：先调用MyType的析构函数，然后调用free释放原先申请的内存。
 - delete只用于删除由new创建的对象。对于用malloc等C标准函数申请的对象，使用delete的行为是未定义的。
 - 如果正在删除的对象是一个指向0地址的指针，delete将不执行任何动作。为此，有时建议在删除指针后立即把指针赋值为0以免对它删除两次。

动态内存分配：从C语言到C++

- 比较：
 - 创建对象通常不仅仅是一个获取内存的过程，通常对象的关键信息需要在构造的时候初始化。面向对象程序设计语言中将这两个步骤联系在一起可以防止很多因遗漏初始化而引发的程序错误。
 - 同样，删除对象也不仅仅是一个释放内存的过程。

动态内存分配： 示例

```
class Tree {  
    int height;  
public:  
    Tree(int treeHeight) : height(treeHeight) {}  
    ~Tree() { std::cout << "*"; }  
    friend std::ostream&  
    operator<<(std::ostream& os, const Tree* t) {  
        return os << "Tree height is: "  
                << t->height << std::endl;  
    }  
  
};
```

C13:NewAndDelete.cpp

```
int main() {  
    Tree* t = new Tree(40);  
    cout << t;  
    delete t;  
} ///:~
```

内存管理的开销

- 内存申请的过程
 - 从堆中搜索一块连续的足够大的内存，没有被使用过的内存
 - 在某处标记该内存已经被使用
 - 返回指向该内存的指针
- 自动垃圾回收与效率

delete void*

- 在C++中，任何指针都可以被转换为void*。如果一个用new 申请的对象指针被转换为了void*，在delete时的行为将发生变化。
- C13:BadVoidPointerDeletion.cpp ➔
- 内存泄漏
 - 如果在程序中发现内存丢失的情况，那么就搜索所有的delete语句并检查被删除指针的类型。如果是void*类型，就可能发现了引起内存泄漏的一个因素。

```
class Object {
    void* data; // Some storage
    const int size;
    const char id;
public:
    Object(int sz, char c) : size(sz), id(c) {
        data = new char[size];
        cout << "Constructing object " << id
              << ", size = " << size << endl;
    }
    ~Object() {
        cout << "Destructing object " << id << endl;
        delete []data; // OK, just releases storage,
        // no destructor calls are necessary
    }
};
```

```
int main() {
    Object* a = new Object(40, 'a');
    delete a;
    void* b = new Object(40, 'b');
    delete b;
} ///:~
```

Constructing object a, size = 40 Destructing object a Constructing object b, size = 40
--

释放指针所指向内存的责任

- 通常，谁创建的对象，就由谁来释放对象。
- 资源的申请和释放通常具有三明治结构。
- 回顾：
 - C09: Stash4.h
 - C09: Stash4.cpp
 - C09: Stash4Test.cpp
- 上面的实现在对象生命周期的管理上存在什么问题？

指针的Stash

- 旧Stash的问题主要是：
 - 它只管理了对象的部分内容。
- 作为容器，要么将对象的所有内容都管理起来（使用拷贝构造函数），要么只管理对象的指针。
- C13:PStash.h
- C13:PStash.cpp
- C13:PStashTest.cpp

数组与指针

- 在C/C++中，指针和数组可以相互转换


```
int my_array[] = {1,23,17,4,-5,100};  
int *ptr;
```

```
int main(void)  
{  
    int i;  
    ptr = &my_array[0];    /* point our pointer to the first element of the array */  
    printf("\n\n");  
    for (i = 0; i < 6; i++)  
    {  
        printf("my_array[%d] = %d  ",i,my_array[i]);  /*<-- A */  
        printf("ptr + %d = %d\n",i, *(ptr + i));      /*<-- B */  
    }  
    return 0;  
}
```

数组与指针

- 使用下面的方法可以使得指针更接近于数组
- `int* const q = new int[10]`

用于数组的new和delete

- 创建一个对象数组的语法
 - `MyType* fp = new MyType[100];`
- 创建一个对象数组的过程分为下面的步骤：
 - 在堆上搜索并申请一块连续的，能够放下指定数量对象的内存。
 - 为每个对象调用缺省构造函数。
- 删除一个数组时的语法如下
 - `delete fp;`  有问题吗？
 - `delete []fp;`

用于数组的new和delete

- 使指针更像数组
 - `int const* q = new int[10];` //指针变量指向常量;
 - `const int* q = new int[10];`
 - `int* const q = new int[10];`

耗尽内存

- 在new无法申请到内存时，将调用一个处理函数。[C13:NewHandler.cpp]

```
int count = 0;

void out_of_memory() {
    cerr << "memory exhausted after " << count
        << " allocations!" << endl;
    exit(1);
}

int main() {
    set_new_handler(out_of_memory);
    while(1) {
        count++;
        new int[1000]; // Exhausts memory
    }
} ///:~
```

重载new 和 delete

- 使用new和delete进行内存分配是为通用的目的而设计的。但在特殊的情形下，可能不满足需要。
- 最常见的改变分配内存方式的原因是出于效率的考虑：
 - 性能：需要创建和销毁一个特定的类的非常多的对象，以至于这个运算成了速度的瓶颈。
 - 堆碎片：频繁分配和释放不同大小的内存可能会导致内存碎片。
- 在嵌入式系统这类内存受限的环境，可能需要定制内存分配策略。

重载new 和 delete

- 使用new和delete申请和释放对象包括内存的申请和对象的初始化，或者内存的释放和对象的清理。
- 对new和delete的重载只能够改变内存的申请和释放。编译器在看到new时，总是会调用对象的构造函数和析构函数。
- 重载new运算符时也可以替换内存耗尽时的行为。

重载new 和 delete的理由

- 重写new和delete的8个理由：
 - 1) To detect usage errors.
 - 2) To improve efficiency.
 - 3) To collect usage statistics.
 - 4) To increase the speed of allocation and deallocation.
 - 5) To reduce the space overhead of default memory management.
 - 6) To compensate for suboptimal alignment in the default allocator.
 - 7) To cluster related objects near one another.
 - 8) To obtain unconventional behavior.

重载全局new 和 delete运算符

- 全局重载
 - 当全局版本的new和delete不能满足系统的需求时，可以对其重载，这将使得默认的版本不能访问。
 - new
 - 参数： size_t
 - 返回： void*
 - delete
 - 参数： void*
 - 返回： void
- 示例
 - C13:GlobalOperatorNew.cpp

重载全局new 和 delete运算符

- C13:GlobalOperatorNew.cpp

```
void* operator new(size_t sz) {  
    printf("operator new: %d Bytes\n", sz);  
    void* m = malloc(sz);  
    if(!m) puts("out of memory");  
    return m;  
}  
  
void operator delete(void* m) {  
    puts("operator delete");  
    free(m);  
}
```


重载类new和delete

- 可以定义针对特定类的new和delete操作符。
- 为一个类重载new和delete时，尽管不必显式地使用static，但实际上仍是在创建static成员函数
 - 非static的new有意义吗？
- 示例：
 - C13:Framis.cpp

定位new和delete

- 我们可能需要在内存的指定位置上放置一个对象。这对于面向硬件的嵌入式系统特别重要，在这个系统中，一个对象可能和一个特定的硬件是同义的。
- 可以给new带除了对象构造参数以外的额外的参数。

```
X* xp = new(a) X;
```

定位new和delete

```
class X {
    int i;
public:
    X(int ii = 0) : i(ii) {
        cout << "this = " << this << endl;
    }
    ~X() {
        cout << "X::~~X(): " << this << endl;
    }
    void* operator new(size_t, void* loc) {
        return loc;
    }
};

int main() {
    int l[10];
    cout << "l = " << l << endl;
    X* xp = new(l) X(47); // X at location l
    xp->X::~~X(); // Explicit destructor call
    // ONLY use with placement!
} ///:~
```

重载数组new和delete

- 如果为一个类重载了new和delete，那么无论何时创建这个类的一个实例，编译器都将调用这些运算符。但如果要创建这个类的一个对象数组时，全局的new会被调用，这不是我们期望的行为。
- 为了控制数组中对象的构造，需要重载这两个操作符的数组版本。
- 示例：
 - C13:ArrayOperatorNew.cpp

小结

- 使用new创建和初始化对象
- 使用delete清理和删除对象
- 重新定义new/delete操作符